

# 浙江大学

## 本科实验报告

课程名称: 操作系统

姓 名:

学 院: 计算机科学与技术学院

系: 计算机科学与技术系

专 业: 计算机科学与技术

学 号:

指导教师: 李环

2024 年 12 月 9 日

# 浙江大学操作系统实验报告

实验名称：\_\_\_\_\_RV64 缺页异常处理与 fork 机制\_\_\_\_\_

电子邮件地址：\_\_\_\_\_手机：\_\_\_\_\_

实验地点：\_\_\_\_\_曹西 503\_\_\_\_\_实验日期：2024 年 12 月 9 日

## 一、实验目的和要求

通过 `vm_area_struct` 数据结构实现对进程多区域虚拟内存的管理

在 Lab4 实现用户态程序的基础上，添加缺页异常处理 `page fault handler`

为进程加入 `fork` 机制，能够支持通过 `fork` 创建新的用户态进程

## 二、实验过程

### （一）准备工程

从本仓库同步 `user/main.c`，并删除原来的 `getpid.c`：

修改 `user` 目录下的 `Makefile`

1. `TEST` = `PFH1`
- 2.
3. `CFLAG` = ...末尾添加 `-D$(TEST)`

### （二）缺页异常处理

#### 1. 实现虚拟内存管理功能

每块 `vma` 都有自己的 `flag` 来定义权限以及分类（是否匿名），在适当的地方添加以下宏定义：

1. `#define VM_ANON 0x1`
2. `#define VM_READ 0x2`

```

3.  #define VM_WRITE 0x4
4.  #define VM_EXEC 0x8

```

接下来要添加 vma 的数据结构，我们采用链表的实现：

```

1.  struct vm_area_struct {
2.      struct mm_struct *vm_mm;    // 所属的 mm_struct
3.      uint64_t vm_start;          // VMA 对应的用户态虚拟地址的开始
4.      uint64_t vm_end;            // VMA 对应的用户态虚拟地址的结束
5.      struct vm_area_struct *vm_next, *vm_prev; // 链表指针
6.      uint64_t vm_flags;          // VMA 对应的 flags
7.      // struct file *vm_file;    // 对应的文件（目前还没实现，而且我们
    // 只有一个 uapp 所以暂不需要）
8.      uint64_t vm_pgoff;          // 如果对应了一个文件，那么这
    // 块 VMA 起始地址对应的文件内容相对文件起始位置的偏移量
9.      uint64_t vm_filesz;        // 对应的文件内容的长度
10. };
11.
12. struct mm_struct {
13.     struct vm_area_struct *mmap;
14. };
15.
16. struct task_struct {
17.     uint64_t state;
18.     uint64_t counter;
19.     uint64_t priority;
20.     uint64_t pid;
21.
22.     struct thread_struct thread;
23.     uint64_t *pgd;
24.     struct mm_struct mm;
25. };

```

为了支持 demand paging，我们需要支持对 vm\_area\_struct 的添加和查找：

find\_vma 函数：实现对 vm\_area\_struct 的查找

根据传入的地址 addr，遍历链表 mm 包含的 VMA 链表，找到该地址所在的 vm\_area\_struct

如果链表中所有的 vm\_area\_struct 都不包含该地址，则返回 NULL

```

1.  struct vm_area_struct *find_vma(struct mm_struct *mm, uint64_t ad
    dr) {
2.      struct vm_area_struct *vma;

```

```

3.         for (vma = mm->mmap; vma != NULL; vma = vma->vm_next) {
4.             if (vma->vm_start <= addr && vma->vm_end >= addr) {
5.                 return vma;
6.             }
7.         }
8.         return NULL;
9.     }

```

`do_mmap` 函数：实现 `vm_area_struct` 的添加

新建 `vm_area_struct` 结构体，根据传入的参数对结构体赋值，并添加到 `mm` 指向的 VMA 链表中

```

1.  void do_mmap(struct mm_struct *mm, uint64_t addr, uint64_t len, u
   int64_t vm_pgoff, uint64_t vm_filesz, uint64_t flags) {
2.     struct vm_area_struct *temp = (struct vm_area_struct *)kalloc
   ();
3.     temp->vm_start = addr;
4.     temp->vm_end = addr + len;
5.     temp->vm_flags = flags;
6.     temp->vm_pgoff = vm_pgoff;
7.     temp->vm_filesz = vm_filesz;
8.     temp->vm_next = NULL;
9.     temp->vm_mm = mm;
10.    struct vm_area_struct *vma = mm->mmap;
11.    if (vma == NULL) {
12.        mm->mmap = temp;
13.        temp->vm_prev = NULL;
14.        return;
15.    }
16.    while (vma->vm_next != NULL) {
17.        vma = vma->vm_next;
18.    }
19.    vma->vm_next = temp;
20.    temp->vm_prev = vma;
21.    return;
22. }

```

## 2. 修改 `task_init()`

接下来我们要修改 `task_init` 来实现 demand paging。

例如，我们原本要为用户态虚拟地址映射一个页，需要进行如下操作：

使用 `kalloc` 或者 `alloc_page` 分配一个页的空间

对这个页中的数据进行填充

将这个页映射到用户空间，供用户程序访问。并设置好对应的 `U, W, X, R` 权限，最后将 `V` 置为 `1`，代表其有效。

而为了减少 `task` 初始化时的开销，我们这样对一个 `Segment` 或者用户态的栈建立映射的操作只需改成分别建立一个 `VMA` 即可，具体的分配空间、填充数据的操作等后面再来完成。

所以我们需要修改 `task_init` 函数代码，更改为 `demand paging`：

删除（注释）掉之前实验中对用户栈、代码 `load segment` 的映射操作（`alloc` 和 `create_mapping`）

调用 `do_mmap` 函数，建立用户 `task` 的虚拟地址空间信息，在本次实验中仅包括两个区域：代码和数据区域：该区域从 ELF 给出的 `Segment` 起始用户态虚拟地址 `phdr->p_vaddr` 开始，对应文件中偏移量为 `phdr->p_offset` 开始的部分

用户栈：范围为 `[USER_END - PGSIZE, USER_END)`，权限为 `VM_READ | VM_WRITE`，并且是匿名的区域（`VM_ANON`）

```
1.  static uint64_t load_program(struct task_struct *task) {
2.      Elf64_Ehdr *ehdr = (Elf64_Ehdr *)_sramdisk;
3.      Elf64_Phdr *phdrs = (Elf64_Phdr *)(_sramdisk + ehdr->e_phoff)
4.      ;
5.      for (int i = 0; i < ehdr->e_phnum; ++i) {
6.          Elf64_Phdr *phdr = phdrs + i;
7.          if (phdr->p_type == PT_LOAD) {
8.              /*uint64_t offset = (uint64_t)(phdr->p_vaddr)-PGROUND
9.              DOWN(phdr->p_vaddr);
10.             uint64_t size = PGROUNDUP(phdr->p_memsz + offset) / P
11.             GSIZE;
12.             uint64_t ad = alloc_pages(size);
13.             uint64_t src_start = (uint64_t)_sramdisk + phdr->p_of
14.             fset;
15.             for (int j = 0; j < phdr->p_memsz; ++j) {
16.                 *((char*)ad + j + offset) = *((char*)src_start +
17.                 j);
18.             }
19.             memset((void*)(ad + phdr->p_filesz + offset), 0, phdr
20.             ->p_memsz-phdr->p_filesz);*/
21.             uint64_t perm = 0;
22.             perm |= (phdr->p_flags & 1) ? VM_EXEC : 0; //X
```

```

17.         perm |= (phdr->p_flags & 2) ? VM_WRITE : 0; //W
18.         perm |= (phdr->p_flags & 4) ? VM_READ : 0; //R
19.         //create_mapping(task->pgd, PGROUNDDOWN(phdr->p_vaddr)
        , ad - PA2VA_OFFSET, phdr->p_memsz + offset, perm);
20.         //printfk(" %lx, %lx", phdr->p_vaddr, phdr->p_vaddr + phd
        r->p_memsz);
21.         do_mmap(&(task->mm), phdr->p_vaddr, phdr->p_memsz, ph
        dr->p_offset, phdr->p_filesz, perm);
22.         // alloc space and copy content
23.         // do mapping
24.         // code...
25.     }
26. }
27. //uint64_t U_stack_top = alloc_page();
28. //create_mapping(task->pgd, USER_END - PGSIZE, U_stack_top -
        PA2VA_OFFSET, PGSIZE, 23);
29. task->thread.sepc = ehdr->e_entry;
30. task->thread.sstatus = csr_read(sstatus);
31. task->thread.sstatus &= ~(1 << 8);
32. task->thread.sstatus |= (1 << 5);
33. task->thread.sstatus |= (1 << 18);
34. task->thread.sscratch = USER_END;
35. return ehdr->e_entry;
36. }

```

```

1. void task_init() {
2.     srand(2024);
3.
4.     // 1. 调用 kalloc() 为 idle 分配一个物理页
5.     // 2. 设置 state 为 TASK_RUNNING;
6.     // 3. 由于 idle 不参与调度, 可以将其 counter / priority 设置为 0
7.     // 4. 设置 idle 的 pid 为 0
8.     // 5. 将 current 和 task[0] 指向 idle
9.
10.    idle = (struct task_struct*) kalloc();
11.    idle->state = TASK_RUNNING;
12.    idle->counter = 0;
13.    idle->priority = 0;
14.    idle->pid = 0;
15.    current = idle;
16.    task[0] = idle;
17.
18.    // 1. 参考 idle 的设置, 为 task[1] ~ task[NR_TASKS - 1] 进行初始
        化

```

```

19.      // 2. 其中每个线程的 state 为 TASK_RUNNING, 此外,
      counter 和 priority 进行如下赋值:
20.      //      - counter = 0;
21.      //      - priority = rand() 产生的随机数 (控制范围
      在 [PRIORITY_MIN, PRIORITY_MAX] 之间)
22.      // 3. 为 task[1] ~ task[NR_TASKS - 1] 设置 thread_struct 中
      的 ra 和 sp
23.      //      - ra 设置为 __dummy (见 4.2.2) 的地址
24.      //      - sp 设置为该线程申请的物理页的高地址
25.
26.      for (int i = 1; i < NR_TASKS; i++){
27.          task[i] = (struct task_struct*) kalloc();
28.          task[i]->state = TASK_RUNNING;
29.          task[i]->counter = 0;
30.          task[i]->priority = rand() % (PRIORITY_MAX - PRIORITY_MIN
      + 1) + PRIORITY_MIN;
31.          task[i]->pid = i;
32.          task[i]->thread.ra = (uint64_t) &__dummy;
33.          task[i]->thread.sp = (uint64_t) task[i] + PGSIZE;
34.          /*struct mm_struct *mm = (struct mm_struct *)kalloc();
35.          mm->mmap = NULL;
36.          task[i]->mm = *mm;*/
37.          task[i]->mm.mmap = NULL;
38.
39.          /*task[i]->thread.sepc = USER_START;
40.          task[i]->thread.sstatus = csr_read(sstatus);
41.          task[i]->thread.sstatus &= ~(1 << 8); //sret 返回 u-mode
42.          task[i]->thread.sstatus |= (1 << 5); //sret 开启中断
43.          task[i]->thread.sstatus |= (1 << 18); //s-mode 可以访问用户
      态进程页面
44.          task[i]->thread.sscratch = USER_END;*/
45.
46.          task[i]->pgd = (uint64_t)alloc_page();
47.          for (int j = 0; j < 512; j++){
48.              task[i]->pgd[j] = swapper_pg_dir[j];
49.          }
50.          do_mmap(&(task[i]->mm), USER_END-PGSIZE, PGSIZE, 0, 0, VM
      _READ | VM_WRITE | VM_ANON);
51.          task[i]->thread.sepc = load_program(task[i]);
52.
53.          /*uint64_t U_stack_top = kalloc();
54.
55.          uint64_t size = PGROUNDUP((uint64_t)_eramdisk - (uint64_t)
      _sramdisk) / PGSIZE;

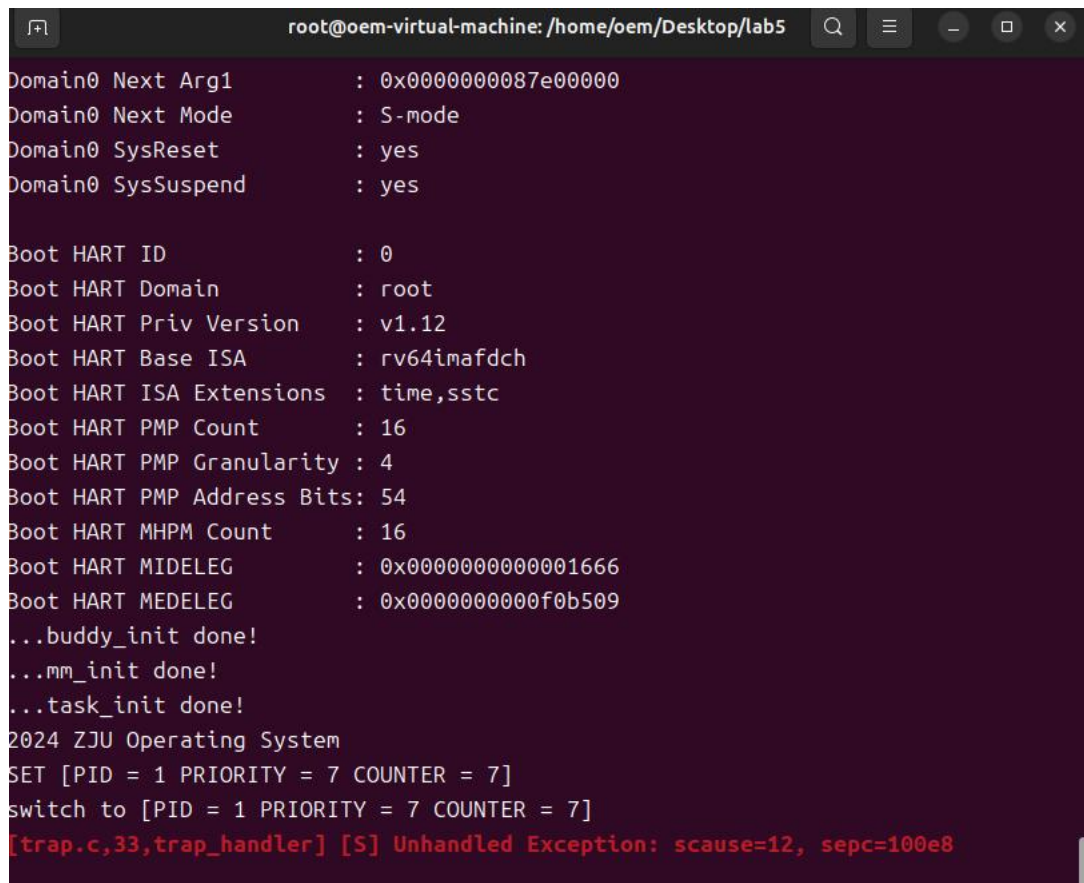
```

```

56.         uint64_t copy_addr = alloc_pages(size);
57.         for (int j = 0; j < _eramdisk - _sramdisk; ++j)
58.             ((char *)copy_addr)[j] = _sramdisk[j];
59.
60.         create_mapping(task[i]->pgd, USER_START, (uint64_t)copy_a
            ddr - PA2VA_OFFSET, size * PGSIZE, 31); // 映射用户段 U|X|W|R|V
61.         create_mapping(task[i]->pgd, USER_END - PGSIZE, U_stack_t
            op - PA2VA_OFFSET, PGSIZE, 23); // 映射用户栈 U|-|W|R|V*/
62.     }
63.
64.
65.     printk("...task_init done!\n");
66. }

```

在完成上述修改之后，如果运行代码我们就可以截获一个 page fault，如下所示：



```

root@oem-virtual-machine: /home/oem/Desktop/lab5
Domain0 Next Arg1      : 0x00000000087e0000
Domain0 Next Mode      : S-mode
Domain0 SysReset       : yes
Domain0 SysSuspend     : yes

Boot HART ID           : 0
Boot HART Domain       : root
Boot HART Priv Version  : v1.12
Boot HART Base ISA     : rv64imafdc
Boot HART ISA Extensions : time,sstc
Boot HART PMP Count    : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count   : 16
Boot HART MIDELEG       : 0x00000000000001666
Boot HART MEDELEG      : 0x00000000000f0b509
...buddy_init done!
...mm_init done!
...task_init done!
2024 ZJU Operating System
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[trap.c,33,trap_handler] [S] Unhandled Exception: scause=12, sepc=100e8

```

### 3. 实现 page fault handler

接下来我们需要修改 trap.c, 为 trap\_handler 添加捕获 page fault 的逻辑, 分别需要捕获 12, 13, 15 号异常。



当捕获了 `page fault` 之后，需要实现缺页异常的处理函数 `do_page_fault`，它可以同时处理三种不同的 `page fault`。

函数的具体逻辑为：

通过 `stval` 获得访问出错的虚拟内存地址（Bad Address）

通过 `find_vma()` 查找 `bad address` 是否在某个 `vma` 中

如果不在，则出现非预期错误，可以通过 `Err` 宏输出错误信息

如果在，则根据 `vma` 的 `flags` 权限判断当前 `page fault` 是否合法

如果非法（比如触发的是 `instruction page fault` 但 `vma` 权限不允许执行），则 `Err` 输出错误信息

其他情况合法，需要我们按接下来的流程创建映射

分配一个页，接下来要将这个页映射到对应的用户地址空间

通过 `(vma->vm_flags & VM_ANON)` 获得当前的 `VMA` 是否是匿名空间

如果是匿名空间，则直接映射即可

如果不是，则需要根据 `vma->vm_pgoff` 等信息从 `ELF` 中读取数据，填充后映射到用户空间

```
1. void do_page_fault(struct pt_regs *regs) {
2.     uint64_t stval = csr_read(stval);
3.     struct vm_area_struct *pgf_vm_area = find_vma(&(current->mm),
        stval);
4.     if (pgf_vm_area == NULL){
5.         Err("illegal page fault");
6.     }
7.     uint64_t va = PGROUNDDOWN(stval);
8.     uint64_t sz = PGROUNDUP(stval + PGSIZE) - va;
9.     uint64_t pa = alloc_pages(sz / PGSIZE);
10.    uint64_t perm = !(pgf_vm_area->vm_flags & VM_READ) * PTE_R |
11.        !(pgf_vm_area->vm_flags & VM_WRITE) * PTE_W
12.        |
13.        !(pgf_vm_area->vm_flags & VM_EXEC) * PTE_X |
14.        PTE_U | PTE_V;
15.    memset((void*)pa, 0, sz);
16.    if (pgf_vm_area->vm_flags & VM_ANON) {
17.
18.    } else {
19.        uint64_t src = (uint64_t)_sramdisk + pgf_vm_area->vm_pgoff;
        f;
```

```

20.         uint64_t offset = stval - pgf_vm_area->vm_start;
21.         uint64_t src_uapp = PGROUNDDOWN(src + offset);
22.         for (int j = 0; j < sz; j++) {
23.             ((char*)(pa))[j] = ((char*)src_uapp)[j];
24.         }
25.     }
26.
27.     create_mapping(current->pgd, va, pa-PA2VA_OFFSET, sz, perm);
28. }

```

### （三）实现 fork 系统调用

在实现较为复杂的 fork 流程之前，我们先将框架搭好，具体要做的有以下两件事：

修改 proc 相关代码，使其只初始化一个进程，其他进程保留为 NULL 等待 fork 创建

添加系统调用处理：在系统调用的处理函数中，检测到 regs->a7 == SYS\_CLONE 时，调用

do\_fork 函数来完成 fork 的工作

我们可以梳理一下 fork 的工作：

创建一个新进程：

拷贝内核栈（包括了 task\_struct 等信息）

创建一个新的页表

拷贝内核页表 swapper\_pg\_dir

遍历父进程 vma，并遍历父进程页表

将这个 vma 也添加到新进程的 vma 链表中

如果该 vma 项有对应的页表项存在（说明已经创建了映射），则需要深拷贝一整页的内容

并映射到新页表中

将新进程加入调度队列

处理父子进程的返回值

父进程通过 do\_fork 函数直接返回子进程的 pid，并回到自身运行

子进程通过被调度器调度后（跳到 thread.ra），开始执行并返回 0

```

1.     uint64_t sys_clone(struct pt_regs* regs) {
2.         int pid = 0;
3.         while (pid < NR_TASKS) {
4.             if (!task[pid]) {
5.                 break;
6.             }

```

```

7.         ++pid;
8.     }
9.     if (pid >= NR_TASKS) {
10.         return -1;
11.     }
12.
13.     struct task_struct* child_task;
14.     child_task = (struct task_struct*)kalloc();
15.     task[pid] = child_task;
16.     for (int i = 0; i < 1<<12; i++) {
17.         ((char*)child_task)[i] = ((char*)current)[i];
18.     }
19.     child_task->pid = pid;
20.     child_task->thread.ra = (uint64_t)__ret_from_fork;
21.
22.     uint64_t offset = (uint64_t)regs - (uint64_t)current;
23.     struct pt_regs *child_regs = (struct pt_regs *)((uint64_t)child_task + offset);
24.     child_regs->x[10] = 0;
25.     child_regs->x[2] = (uint64_t)child_regs;
26.     child_regs->sepc = regs->sepc + 4;
27.     child_task->thread.sp = (uint64_t)child_regs;
28.     child_task->thread.sscratch = csr_read(sscratch); //(uint64_t)
        child_regs;
29.
30.     child_task->pgd = (uint64_t)alloc_page();
31.     memset(child_task->pgd, 0, 1 << 12);
32.     for (int j = 0; j < 512; j++) {
33.         child_task->pgd[j] = swapper_pg_dir[j];
34.     }
35.     child_task->mm.mmap = NULL;
36.
37.     for (struct vm_area_struct *vma = current->mm.mmap; vma != NULL; vma = vma->vm_next) {
38.         do_mmap(&(child_task->mm), vma->vm_start, vma->vm_end - vma->vm_start, vma->vm_pgoff, vma->vm_filesz, vma->vm_flags);
39.         for (uint64_t vaddr = PGROUNDDOWN(vma->vm_start); vaddr < vma->vm_end; vaddr += PGSIZE) {
40.             if ((current->pgd[(vaddr >> 30) & 0x1fff] & (PTE_V)) = PTE_V) {
41.                 uint64_t child_page = alloc_page();
42.                 for (int j = 0; j < PGSIZE; j++) {
43.                     ((char *)child_page)[j] = ((char *)vaddr)[j];

```

```

44.             create_mapping(child_task->pgd, vaddr, child_
    page-PA2VA_OFFSET, PGSIZE, vma->vm_flags | PTE_U | PTE_V);
45.             }
46.         }
47.     }
48. }
49.     printk("[PID = %d] forked from [PID = %d]\n",child_task->pid,
    current->pid);
50.     return child_task->pid;
51. }

```

利用 `__switch_to` 时恢复的 `ra` 与 `sp`，我们可以让子进程返回时直接跳转到 `_traps` 中从 `trap_handler` 返回的位置，只需要加一个标号：

```

1.     _traps:
2.         ...
3.         jal trap_handler
4.
5.         .globl __ret_from_fork
6.     __ret_from_fork:
7.
8.         ...

```

#### （四）测试 fork

测试结果如下（FORK1）：

```

...buddy_init done!
...mm_init done!
...task_init done!
2024 ZJU Operating System
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[PID = 2] forked from [PID = 1]
[U-PARENT] pid: 1 is running! global_variable: 0
[U-PARENT] pid: 1 is running! global_variable: 1
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[U-CHILD] pid: 2 is running! global_variable: 0
[U-CHILD] pid: 2 is running! global_variable: 1
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-PARENT] pid: 1 is running! global_variable: 2
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[U-CHILD] pid: 2 is running! global_variable: 2
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-PARENT] pid: 1 is running! global_variable: 3
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]

```

测试结果如下（FORK2）：

```

...buddy_init done!
...mm_init done!
...task_init done!
2024 ZJU Operating System
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U] pid: 1 is running! global_variable: 0
[U] pid: 1 is running! global_variable: 1
[U] pid: 1 is running! global_variable: 2
[PID = 2] forked from [PID = 1]
[U-PARENT] pid: 1 is running! Message: ZJU OS Lab5
[U-PARENT] pid: 1 is running! global_variable: 3
[U-PARENT] pid: 1 is running! global_variable: 4
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[U-CHILD] pid: 2 is running! Message: ZJU OS Lab5
[U-CHILD] pid: 2 is running! global_variable: 3
[U-CHILD] pid: 2 is running! global_variable: 4
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U-PARENT] pid: 1 is running! global_variable: 5
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[U-CHILD] pid: 2 is running! global variable: 5

```

测试结果如下（FORK3）：

```
2024 ZJU Operating System
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[U] pid: 1 is running! global_variable: 0
[PID = 2] forked from [PID = 1]
[PID = 3] forked from [PID = 1]
[U] pid: 1 is running! global_variable: 1
[PID = 4] forked from [PID = 1]
[U] pid: 1 is running! global_variable: 2
[U] pid: 1 is running! global_variable: 3
switch to [PID = 2 PRIORITY = 7 COUNTER = 7]
[U] pid: 2 is running! global_variable: 1
[U] pid: 2 is running! global_variable: 2
[U] pid: 2 is running! global_variable: 3
switch to [PID = 3 PRIORITY = 7 COUNTER = 7]
[U] pid: 3 is running! global_variable: 1
[U] pid: 3 is running! global_variable: 2
[U] pid: 3 is running! global_variable: 3
switch to [PID = 4 PRIORITY = 7 COUNTER = 7]
[U] pid: 4 is running! global_variable: 2
[U] pid: 4 is running! global_variable: 3
SET [PID = 1 PRIORITY = 7 COUNTER = 7]
SET [PID = 2 PRIORITY = 7 COUNTER = 7]
SET [PID = 3 PRIORITY = 7 COUNTER = 7]
```

根据图中显示，测试结果正常。

## 三、讨论和心得

本次实验建立在 lab4 环境已经搭建好的基础上，逻辑上更为复杂，需要理解各种函数的逻辑以及各种指针的值应该赋什么。在本次实验中，有一点需要注意，在 entry.S 的 `_trap` 函数中需要额外保存 `stval`, `sscratch`, `scause` 这三个寄存器，如果不保存程序就会出现未知的缺页异常，而这正是实验指导里没有说的，我在这个地方卡了好久。

这次实验让我对缺页异常的工作机制有了更深入的了解，同时也加深了对用户态进程 `fork` 的理解。

## 四、思考题

1.

`do_page_fault` 要先进行 `bad address` 的检查，如果在分配的范围内再创建映射，不能分配的话（不在范围内或者权限不对）结束进程。首先分配一个页，判断当前的 `VMA` 是否为匿名



空间，如果不是匿名空间，从 ELF 中读取数据，填充数据的起始地址为 `_sramdisk + pgf_vm_area->vm_pgoff + stval - pgf_vm_area->vm_start`，再将数据复制到以这个页地址为起始地址的连续空间上，再做 `create_mapping` 如果是匿名空间直接 `create_mapping` 即可。

2.

父进程内核栈的指针是 `thread.sp`，用户栈的指针是 `thread.sscratch`，子进程的内核栈指针的值应该是父进程内核栈指针地址减去父进程结构体指针地址加上子进程结构地址，即拥有相同的 `offset`，子进程用户栈指针的值是 `csr_read(sscratch)`，即父进程保存在 `sscratch` 寄存器里的值，内核栈指针应该赋值给子进程的 `thread.sp` 以及子进程 `pt_reg` 的 `x[2]` 位置，子进程的用户栈指针应该赋值给子进程的 `thread.sscratch`。

3.

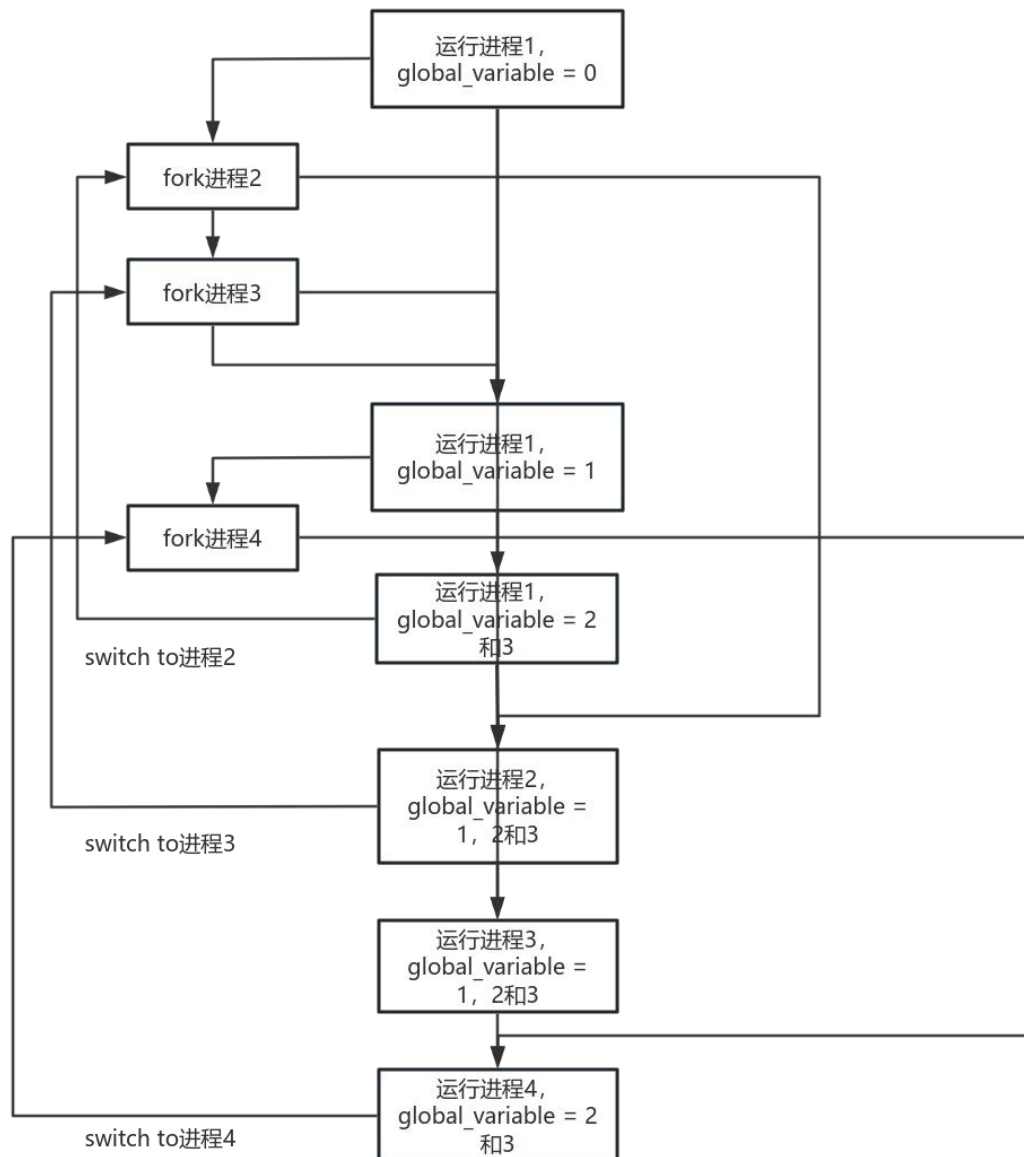
异常发生时，`sepc` 会保存触发异常的指令地址，当 `fork` 指令触发 `ecall` 时，`sepc` 指向的将是 `ecall` 指令的地址，`ecall` 指令在执行时会停止当前程序的执行，从用户态进入内核态，因此，处理完系统调用后，`sepc` 需要加 4，使得程序从系统调用返回时继续执行 `fork` 之后的下一条指令，如果不加 4 会再回到同一位置触发无限 `fork` 导致死循环。

4.

ZJU OS Lab5 位于内存的用户栈上。运行时产生了 `scause` 为 15 的 `store page fault`。在 `pid=1` 的进程运行完前 3 次后，程序捕捉到 `scause` 为 15 的缺页异常，如图所示。

```
[U] pid: 1 is running! global_variable: 0
[U] pid: 1 is running! global_variable: 1
[U] pid: 1 is running! global_variable: 2
scause = 15, sepc = 1023c, stval = 13008
[PID = 2] forked from [PID = 1]
[U-PARENT] pid: 1 is running! Message: ZJU OS Lab5
[U-PARENT] pid: 1 is running! global_variable: 3
[U-PARENT] pid: 1 is running! global_variable: 4
```

5.



进程 2 和 3 的 global\_variable 应该从 1 开始输出，进程 4 的 global\_variable 应该从 2 开始输出，与运行结果一致。

## 五、附录

无